VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KONFIGURACE TESTŮ V PROJEKTU APACHE CAMEL POMOCÍ ANOTACÍ

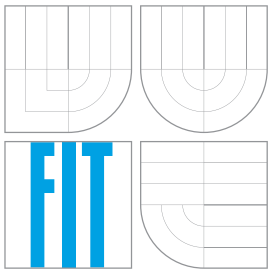
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JOSEF KARÁSEK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KONFIGURACE TESTŮ V PROJEKTU APACHE CAMEL POMOCÍ ANOTACÍ

SUPPORT FOR ANNOTATION DRIVEN TESTING IN APACHE CAMEL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JOSEF KARÁSEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2015

Abstrakt

Tato práce se zabývá převodem imperativního způsobu konfigurace testů v projektu Apache Camel na deklarativní způsob v podobě anotací. Je zde představen projekt Apache Camel s důrazem na jeho komponentu sloužící k testování distribuovaných aplikací, které používají knihovny tohoto projektu. Testovací komponenta je analyzována a na základě vybraných metod jsou navrženy a implementovány anotace. Projekty JBoss Arquillian a Pax Exam slouží jako inspirace pro následující práci.

Abstract

This thesis project transforms the imperative test configuration in the Apache Camel framework in to a declarative configuration using Java annotations. It introduces the testing component of the Apache Camel framework. The testing component is analysed and new annotations are developed based on the selected methods. Sources of inspiration for this thesis include the JBoss Arquillian and Pax Exam projects.

Klíčová slova

anotace, apache camel, java, junit, arquillian, pax exam

Keywords

annotation, apache camel, java, junit, arquillian, pax exam

Citace

Josef Karásek: Konfigurace testů v projektu Apache
Camel pomocí anotací, bakalářská práce, Brno, FIT VUT v Brně, 2015

Konfigurace testů v projektu Apache Camel pomocí anotací

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. a pana Ing. Jiřího Pechance.

.....
Josef Karásek
19. května 2015

Poděkování

Rád bych poděkoval panu Ing. Aleši Smrčkovi, Ph.D za vedení a cenné rady, díky kterým je tento dokument mnohem kvalitnější a panu Ing. Jiřímu Pechancovi za technickou pomoc a zpětnou vazbu na mé zdrojové kódy.

Obsah

Kapitola 1

Úvod

Tato bakalářská práce se zabývá transformací imperativního kódu v testovací komponenetě projektu Apache Camel na deklarativní vyjádření v podobě anotací. Je zde analyzována testovánívací komponenta projektu Apache Camel a vhodné metody (tedy imperativní kód) jsou vybrány pro převod na anotace. Součástí této práce je i implementace nových anotací.

Existuje několik projektů postavených na platformě Java, které se stejně jako Apache Camel zaměřují na testování distribuovaných aplikací. Proto jsou zde představeny i projekty JBoss Arquillian a Pax Exam. Oba projekty definují anotace pro zjednodušení zápisu jednotlivých testovacích případů i konfigurace celé testovací sady a zároveň jsou součástmi větších open-source projektů. Proto jsou zdrojem inspirací pro tuto práci.

Jazyk Java podporuje více programovacích paradigmat zároveň. I když podpora imperativního paradigmatu je nejrozšířenější, umožňuje psát deklarativní kód, a to dvěma způsoby – anotacemi a vytvořením doménově specifického jazyka pomocí běžné syntaxe. Projekt Camel využívá oba přístupy, díky tomu je výsledný kód dobře čitelný a není nepřiměřeně rozsáhlý.

Deklarativní způsob vyjádření v jazyce Java je využíván jak při běhu programu, tak i na úrovní překladu, kde například anotace `@Override` může překladači napovědět, aby byla provedena důkladnější kontrola příslušnosti metody ke třídě a tak již při překladu odhalit možnou sémantickou chybu, která by se jinak projevila až za běhu programu. Knihovny mohou definovat vlastní anotace, za jejich zpracování pak již není zodpovědný kompilátor, ale právě ta knihovna, která anotaci definovala. Tyto anotace jsou vhodné pro inicializaci konfiguračních informací – výsledný kód je kompaktní a přehledný, je zřejmé, která část kódu provádí inicializaci a kde jsou algoritmy provádějící hlavní činnost programu.

Kapitola 2

Testování pomocí anotací v jazyce Java

Před ponořením se do anotací je vhodné říci pár slov o platformě Java. Konkrétně o edici s označením Java Standard Edition, což je souhrnný název pro specifikaci jazyka, specifikaci běhového prostředí a implementace standardních knihoven. V současné době je tato platforma vlastněna společností Oracle, která zároveň poskytuje implementaci zmíněných specifikací. Jak specifikace, tak implementace jsou dostupny pod open source licencemi Sun License a GNU GPL. Díky otevřenosti standardů existuje více implementací, například OpenJDK, která je použita při tvorbě řešení v této bakalářské práci[1]. Samotné specifikace jsou rovněž navrhované odbornou veřejností. Jednotlivec může po bezplatné registraci podat návrh označovaný jako Java Specification Request (společnosti za registraci platí roční poplatek), který bude projednáván komisí. Pokud je schválen, v následující verzi je vydána referenční implementace.

Pro jakoukoli práci s kódem v jazyce Java je nutné nejprve vysvětlit překlad zdrojového kódu, zvláště pak pokud je nutné tento kód (ve skutečnosti jakoukoli jeho podobu, viz dále) zpracovávat. A to pro práci s anotacemi je nutné.

Zdrojový kód jazyka Java není kompilován do strojového kódu, nýbrž pouze do přechodné, binární reprezentace (takzvaný *java bytecode*), který je poté interpretován virtuálním strojem (*Java Virtual Machine*). Výhoda tohoto návrhu je přenositelnost programů. Zdrojové kódy jsou platformově nezávislé, ale implementace *Java Virtual Machine* je závislá na konkrétním hardware a je potřeba vytvořit pro každou platformu specifický virtuální stroj. Nevýhoda návrhu je zpomalení dané interpretací byte kódu oproti exekuci strojového kódu. Zdrojový kód, typicky v souboru s příponou .java je po překladu uchováván v tzv. *class* souboru (s příponou .class). Ten je při spuštění předán virtuálnímu stroji, který jej načte a postupně interpretuje. Tento princip umožnil vznik jiným programovacím jazykům než je Java, které využívají pouze běhového prostředí JVM – překladače těchto jazyků tedy generují java byte kód. Díky společné reprezentaci je možné tyto jazyky propojit, a tak je možné knihovnu napsanou v jazyce Java využívat například v jazyce Scala nebo Groovy.

[1] Použitá verze je 1.8.0_45-b13

2.1 Anotace v jazyce Java

Anotace byly přidány ve verzi Java2SE 5.0 se dvěma záměry:

1. Zakomponování obsahu konfiguračních souborů do zdrojového kódu.
2. Automatické generování jednotvárného kódu (anglicky *boilerplate code*). (Například automatické generování rozhraní dle anotací v implementující třídě.[9])

Anotace v jazyce Java jsou uvozeny prefixem @ (znak zavináč) a nejčastěji jsou použity pro dekoraci tříd, metod a proměnných. Lze je také použít pro dekoraci jiných anotací, balíků (obdoba jmenných prostorů), výčtových typů, rozhraní nebo vymezení na anotaci pouze konstruktoru.

Anotace mají v Javě význam metadat, která identifikují konkrétní procedury a jelikož je zdrojový kód kompilován do byte kódu, pro procedury volané za běhu programu je potřeba tato metadata umět identifikovat právě v byte kódu. Pro anotace identifikující procedury překladače nemá význam zahrnovat je do výstupního byte kódu. Dostáváme se tedy k pojmu vymezení platnosti anotace (anglicky *retention*). Anotace může být součástí:

- zdrojového kódu – Standardní anotace jazyka Java poskytující informace překladači. Pokud chceme vytvořit vlastní anotace na této úrovni, musíme je umět zpracovat při překladu, do výsledného kódu nejsou přidány. K tomu je možné použít *annotation processors*, které jsou součástí překladače od verze Java 6.
- .class souborů – Anotace je součástí .class souborů, ale není dostupná při běhu aplikace. *Java Virtual Machine* je z byte kódu nenačítá.
- byte kódu – Anotace je dostupná při běhu aplikace. Vzniká potřeba ji identifikovat v byte kódu, k tomu je možné použít například standardní knihovnu *Reflection*. Ta je velmi jednoduchá na použití, ale procesovaný byte kód ukládá do paměti jako celek a může tak být pro daný projekt nepřijatelnou paměťovou zátěží. Existují proto knihovny třetích stran manipulující byte kód s menší paměťovou složitostí: JBoss javassist[11] a ASM[10].

Jako demonstrační příklad užití anotace bude prezentována standardní anotace `@Override` jazyka Java dostupná pouze na úrovní zdrojového kódu. Slouží k vynucení silnější kontroly příslušnosti metody ke třídě během překladu – při redefinici metody třídy A ve třídě B je nutné opsat signaturu této metody stejně v obou třídách. Pokud udělá programátor chybu při opisování signatury, neznamená to syntaktickou chybu, nýbrž sémantickou. Původní sémantika „Redefinuji metodu z třídy A ve třídě B“ je změněna na „Ve třídě B definuji novou metodu“. Příklad:

```
public class A {
  public void metodaJedna() {
    // tělo metody
  }
}

public class B extends A {
  @Override
  public void metodajedna() {
    // tělo metody
  }
```

```
}
```

Příklad 2.1: *Anotace @Override*

V uvedeném kódu je zřejmá chyba – názvy metod nesouhlasí (zápis kódu v Javě je case-sensitive[2]). Metoda ve třídě B redefinující metodu třídy A je však opatřená anotací `@Override`, jejíž význam pro překladač znamená „Pokud se v žádné z nadtříd nenalézá metoda se stejnou signaturou, nevytvářej novou metodu, ale ohlaš chybu během překladu.“. Pokud programátor dodržuje tuto konvenci, je při pohledu na kód jasné, která metoda byla deklarována v rodičovské třídě a kde se jedná o první deklaraci. Druhým a mnohem podstatnějším přínosem je odhalení sémantické chyby již během překladu.

Java 5 byla vydána v roce 2004 včetně podpory pro poměrně snadné vytváření vlastních anotací, a tak se anotace rozšířily i do mnoha knihoven, jak standardních v rámci Java Enterprise Edition, tak i mnoha jiných knihoven. Za zpracování takovéto anotace je zodpovědná knihovna, tedy sémantika anotace je dána tvůrci knihovny. Jako příklad bude uvedeno rozhraní pro perzistenciaci objektů pomocí objektově-relačního mapování – Java EE JPA[3]:

```
@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private long id;

  private String name;

  public User() {}  // prázdný konstruktor

  public String getName() {
    return name;
  }

  public void setName(String n) {
    name = n;
  }
}
```

Příklad 2.2: *Anotace v knihovně JPA*

Rozhraní JPA je velice jednoduché na použití. Jediné, co programátor specifikuje je, že vytváří entitu *User*, ta má jako primární klíč proměnnou *id* a atribut *name*. Knihovna JPA pak zajišťuje komunikaci s databází, ve které jsou data perzistentně uchována. Poprvé je uvedena anotace s parametrem, ten v tomto případě specifikuje, že databázový systém se má starat o auto-inkrementaci hodnoty primárního klíče.

Na uvedených příkladech je možné pozorovat deklarativní povahu anotací. Nikde nebylo specifikováno, jak se mají procedury prováděné na pozadí provést. Anotací pouze specifikujeme, že někdy se má něco stát. Jak se to stane je před programátorem skryto a je to součástí vnitřní implementace.

V předchozím příkladu je velmi málo kódu, ale prováděné operace jsou poměrně komplexní. Implementace těchto operací je totiž zapouzdřena v rozhraní JPA, nikde však není žádné přímé volání některé z metod JPA. Jak se tedy JPA dozví, čeho se programátor snaží docílit? Mělo by být zřejmé, že právě z anotací a to z pouhé přítomnosti anotace nebo,

[2] case-sensitive – velká a malá písmena jsou rozlišována.

[3] Java Enterprise Edition, Java Persistence API

pokud je uveden, i z parametru anotace. Rozhraní JPA musí tedy nějakým způsobem tyto anotace najít.

Vytváření anotací

Před vyhledáváním anotace je vhodné vysvětlit, jak taková anotace vypadá. Následuje velmi jednoduchý příklad prezentující vytváření vlastní anotace:

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
  public String value() default "None";
}
```

Příklad 2.3: *Anotace @Author*

Při vytváření anotace se využívají standardní anotace jazyka Java. Do zdrojového kódu byly účelově zahrnuty řádky importující dané anotace – jak je vidět z cesty java.lang.*, součástí standardní knihovny jsou anotace pro vytváření vlastních anotací.

- `@Documented` – Určuje, zda se má anotace zahrnout do dokumentace generované nástrojem *javadoc*[4].
- `@Target` – Seznam konstruktů jazyka Java, které je možno touto anotací dekorovat.
- `@Retention` – Určuje, v jaké fázi překladu/sestavování programu bude anotace odstraněna.
- `@Inherited` – Rozšíření platnosti anotace i na dědící třídy.

Poté následuje deklarace nové anotace, k tomu slouží speciální rozhraní s prefixem @ (zavináč). Název rozhraní je zároveň identifikátor anotace. V těle rozhraní je možné deklarovat parametry anotace (anotace bez parametrů má tělo prázdné). V tuto chvíli je anotace hotova, pro případné parametry platí omezení na primitivní datové typy[5], řetězce a výčtový typ. Klíčovým slovem `default` je možné definovat počáteční hodnotu parametru, v uvedeném příkladu je to řetězec *None*. V případě, že anotace má pouze jeden parametr, je konvencí jej pojmenovat *value*.

Jak lze vyčíst z uvedeného příkladu, anotace `@Author` je možno použít pouze pro dekoraci metod, bude součástí výsledného byte kódu i dokumentace a má jediný parametr. Použití této anotace je uvedeno v příkladu 2.4.

[4] Dokumentace generovaná z komentářů ve zdrojovém kódu.

[5] `byte, short, int, long, float, double, boolean, char`

```
public class AnnotatedMethods {

  @Author("Josef Karasek")
  public void printGreeting(String name) {
      System.out.println("Hello " + name);
  }
}
```

Příklad 2.4: *Použití anotace @Author*

V tuto chvíli však použití této anotace nemá žádný význam – neexistuje žádný nástroj, který by ji zpracovával. Potenciální nástroj zpracovávající uvedenou anotaci ji musí nejdříve nalézt v byte kódu, k čemuž je možné použít knihovnu *Reflection*. Následující příklad načítá byte kód třídy `AnnotatedMethods` a sekvenčně prochází metody (v dané třídě se nachází pouze jedna metoda), pokud některá metoda je anotována anotací `@Author`, je její parametr vypsán na standardní výstup.

```
import java.lang.reflect.Method;

public class RuntimeExample {

  public static void main(String[] args) {
    Class<?> annotatedMethods = AnnotatedMethods.class;

    for (Method m : annotatedMethods.getMethods()) {
      Author annotation = (Author) m.getAnnotation(Author.class);
      if (annotation != null) {
        System.out.println("Author of method " +
                m.getName() + " is " + annotation.value());
      }
    }   // Výstup programu po spuštění bude řetězec:
  }     // Author of method printGreeting is Josef Karasek
}
```

Příklad 2.5: *Práce s knihovnou Reflection*

Co kdyby ale metoda `printGreeting` měla striktnější modifikátor přístupu? *Private* nebo *protected*? Přístup k metodám či instančním proměnným, které programátor úmyslně skryl, může představovat bezpečnostní riziko, nejen že je možné číst stav objektu, ale také jej modifikovat. Pomocí knihovny *Reflection* je možné porušit enkapsulaci a invokovat skrytou metodu, proměnné lze číst i zapisovat. Při porušení zapouzdření pro získání informací o anotaci stav objektu modifikován není a rovněž není invokována žádná privátní metoda. Nicméně při používání knihovny *Reflection* je nutné mít na paměti jisté konvence bezpečného programování[3] a pokud je porušíme, riskujeme zanesení kritické bezpečnostní chyby do naší aplikace. Proto poskytovatelé hostingu pro webové aplikace tuto knihovnu podporují jen v omezené míře[6].

2.2 Testování pomocí JUnit

Tento framework[7] sehrál velmi důležitou roli ve vývoji jednotkového testování a je přímou implementací mnoha poznatků z oblasti agilních vývojových metodik a objektově orientova-

[6] Například Google App Engine nepodporuje žádnou metodu umožňující modifikaci stavu objektu.[5]

[7] Mezi knihovnou a frameworkem jsou zásadní rozdíly – framework je semifunkční aplikace, do které uživatel *zasadí* svůj kód jakožto poslední díl do dokončení aplikace, po spuštění je střídavě vykonáván kód uživatele a frameworku – provádí se tzv. inverze kontroly.

ného návrhu software. Autory JUnit jsou Erich Gamma (*Design Patterns*) a Kent Beck (*Test Driven Development*), který je zároveň autorem méně známého předchůdce JUnit – SUnit, což je první framework zaměřující se na automatizované, jednotkové testování. Cílem SUnit bylo zjednodušit a urychlit jak psaní testů, tak i dobu jejich exekuce a zároveň zdůraznit význam testování v rámci vývojového cyklu software – Beck je jedním ze 16 autorů *agilního manifesta*[8]. Proto vytvořil SUnit jako nástroj pro běžného programátora (tedy nástroj, který nebyl určen specificky pro testery), který zajišťuje jednotné spouštění testů a vyhodnocování výsledků. Zároveň byla sjednocena slovní zásoba specifická pro testování, která je použita i v JUnit. Byl napsán v jazyce Smalltalk a jeho architektura byla založena na čistě objektově orientovaném návrhu.

První verze JUnit vznikla během letu ze Švýcarska do Spojených států, což vypovídá o jednoduchosti tohoto nástroje. Jelikož byl napsán v jazyce Java, byl po vzoru SUnit, kde *S* znamená Smalltalk, pojmenován dle stejného schématu. *Unit* pak značí, že se framework zaměřuje na jednotkové testování, což je testování na velice nízké úrovní z hlediska struktury programu. Zaměřuje se na testování tzv. *jednotky práce*, pod tímto pojmem si lze představit nejčastěji jednu metodu, či krátkou třídu. Kent Beck porovnává jednotkové testování s integračním takto[2]:

1. Kolik cest je v kódu exekuováno? – jednotkový test provede velice malé množství cest v kódu, integrační test naopak mnoho.
2. Pokud test selže, kolik může být v testovaném kódu chyb? – v jednotkovém testu ideálně maximálně jedna a tu je snadné najít. V integračním testu mohlo být naopak odhaleno více chyb, ale je těžší je identifikovat.
3. Kolik času exekuce testu zabere? – jednotkový test by měl proběhnout téměř instantně, je určen k častému spouštění. Na integrační test se tento požadavek však nevztahuje.
4. Kdo čte tyto testy? – jednotkové testy čtou programátoři, kteří implementují danou část aplikace. Integrační testy jsou pak určeny návrhářům aplikace.

Implementace těchto principů v jazyce Java byla mnohem úspěšnější než ta ve Smalltalku a principy se rychle rozšířily do mnoha jiných jazyků: cppUnit(C++), NUnit (C#), phpUnit a mnoho dalších. Principy a slovní zásoba je dnes zastřešena pod názvem xUnit. Základem je princip *setup, exercise, verify, teardown*, tedy příprava[9] prostředí pro spuštění testu, spuštění testované jednotky práce a ověření výsledku, poslední fáze je úklid, tedy transformace prostření do stavu, v jakém bylo před první fází. Slovní zásoba je definována následovně:

1. *Test case* (testovací případ) – třída obsahující metody pro přípravu prostředí, spuštění testovaného kódu a následný úklid.
2. *Test suite* (testovací sada) – množina testovacích případů.
3. *Test fixtures* – metody pro přípravu testovacího prostředí a následný návrat do původního stavu.
4. *Assertion* – metody porovnávající očekávanou hodnotu s hodnotou navrácenou nebo modifikovanou testovaným kódem.

[8] http://agilemanifesto.org

[9] Například vytvoření nebo smazání adresáře, připojení k databázi...

5. *Test runner* – spustitelný program volající metody testovacího případu, po skončení testování vrací výsledky.

Následuje ukázka testu v jazyce Java s použitím JUnit:

```
public class SimpleTest {

  private String s = "not initialized";

  @Before
  public void initializeString() {
    s = "initialized";
  }

  @Test
  public void testStringInitialized() {
    Assert.assertTrue("initialized".equals(s))
  }

  @After
  public void dispose() {
    s = null;
  }
}
```

Příklad 2.6: *Jednoduchý test v JUnit*

Na uvedené ukázce jsou zajímavé dvě věci, JUnit definuje vlastní anotace, kromě nich však v kódu není žádná závislost na frameworku. Testovací třída nedědí od žádné třídy JUnit, ani nevolá žádnou z jeho metod. Toho je docíleno právě pomocí anotací. Jak bylo zmíněno výše, o exekuci testu se stará *test runner*, ten má také na starost identifikaci metod v testovací třídě a rozlišení *test fixtures* metod a testovacích metod, což dělá na základě anotací. Díky tomu je možné psát takto krátký a čitelný kód, ze kterého je chronologie provedení operací jasně čitelná a to jak pro lidi, tak i pro zpracovávající program. Pro úplnost následuje popis událostí testu:

1. Konstrukce objektu třídy `SimpleClass`, inicializace řetězce `s`.
2. Volání metody anotované `@Before`
3. Volání metody anotované `@Test`, pokud je metodě `assertTrue` předán argument hodnoty *false*, je vytvořen objekt výjimky a test je prohlášen za neúspěšný. V opačném případě za úspěšný.
4. Volání metody anotované `@After`

Je nutné uvést, že všechny metody anotované `@Before`, respektive `@After`, budou volány před, respektive po volání každé metody anotované `@Test` (těch může být v testu neomezeně mnoho, vždy však alespoň jedna). K dispozici jsou další dvě anotace: `@BeforeClass` a `@AfterClass`. Metoda anotovaná `@BeforeClass` je volána bezprostředně po vytvoření objektu testované třídy a metoda anotovaná `@AfterClass` před zničením objektu. Každá z nich je tedy volána právě jednou pro daný testovací případ.

Záleží tedy na chronologickém uspořádání operací prováděných během testování, což má na starosti *test runner*. Pokud uživatel uvede více testovacích (nebo *test fixtures*) metod, metody anotované stejnou anotací mají vůči sobě stejnou prioritu. Proces výběru události je

v takovémto případě stochastický, není poskytnuta žádná garance precedence. Tato vlastnost umožňuje použít velice jednoduchý algoritmus řízení testování a není tak třeba speciálních datových struktur, jako je například prioritní fronta. Abstrakce nad použitým algoritmem je stromová hierarchie a spuštění testu znamená průchod tímto stromem. Jelikož není potřeba žádných složitých operací, není tento strom explicitně naprogramován, je použit pouze pro lepší představu o vnitřních mechanizmech JUnit. Podrobněji se tomuto algoritmu věnuje kapitola 3.

Součástí JUnit jsou ještě další anotace, pro účely této práce je potřebné znát již pouze jednu – `@RunWith`. Dá se říci, že se jedná o tu nejdůležitější anotaci pro tuto práci, protože díky ní je možné defaultní *test runner* nahradit vlastním, a tím upravit JUnit dle vlastních potřeb. Toho využívá mnoho frameworků, kdy z JUnit je využit mechanizmus pro spuštění testů a vyhodnocení výsledků, ale je například přidán mechanizmus přípravy testovacího prostředí dle požadavků specifických pro konkrétní framework.

Na závěr je pro získání představy, jak takový unit test vypadá a jaké procedury musí programátor provést, uveden krátký test. Testovaná jednotka práce bude krátká třída, s jedinou metodou. Ta očekává jako argument řetězec a konkatenuje jej s jiným řetězcem. Význam této metody je vytvoření pozdravu.

```
public class Greeter {
  public String createGreeting(String name) {
    return "Hello, " + name + "!";
  }
}
```

Příklad 2.7: *Testovaná třída Greeter*

V testu je nutné vytvořit instanci objektu `Greeter`, zavolat metodu `createGreeting` a porovnat vrácenou hodnotu s hodnotou očekávanou.

```
public class GreeterTest {

  private Greeter greeter;

  @Before
  public void createGreeter() {
    greeter = new Greeter();
  }

  @Test
  public void testCreateGreeting() {
    String expected = "Hello, JUnit!";
    String result = greeter.createGreeting("JUnit");
    Assert.assertEquals(expected, result);
  }
}
```

Příklad 2.8: *JUnit test třídy Greeter*

Objekt `Greeter` je v ukázce uveden jako instanční proměnná testovací třídy. Stejně tak je možné jej definovat jako lokální proměnnou v metodě `testCreateGreeting` a tam jej také inicializovat, namísto v metodě anotované `@Before`. Rozdělení těchto činností však zvyšuje čitelnost testů, metoda provádějící test obsahuje pouze esenciální prostředky pro test, veškerá příprava je v metodě anotované `@Before`.

2.3 Testování pomocí Arquillian

Předchozí kapitola se zaměřuje na jednotkové testování, které je vysvětleno v kontrastu s integračním testováním. Integrační testování bude v této kapitole probráno podrobněji a bude představen projekt Arquillian komunity JBoss, který do JUnit přidává podporu pro integrační testování aplikací běžících na JVM.

Na příkladu uvedeném na konci předchozí kapitoly je vidět operace, které jsou běžně prováděny v unit testu. Integrační test se však věnuje více jednotkám práce, tedy více třídám a metodám. To nemá za následek pouze vytvoření více objektů, ale je zde přítomna další režie. JUnit byl představen jako framework pro testování software, při vývoji aplikace je však často použito mnohem více frameworků a mnoho jiných závislostí. Jak bylo vysvětleno, framework je semifunkční aplikace, tedy implementace velké části logiky je již hotova a je potřeba přidat pouze uživatelem definovanou část. Před spuštěním tedy může probíhat inicializační fáze, kdy se spouští část aplikace definovaná frameworkem a poté se k ní registruje uživatelova část. To samotné je v porovnání s vytvářením malého množství objektů poměrně komplexní proces a pro účely snadného a rychlého integračního testování je potřeba jej automatizovat.

V této kapitole se bude často vyskytovat pojem kontejner, tím je myšleno prostředí, do kterého je registrována uživatelem definovaná část aplikace. Typickým příkladem je aplikační server, který poskytuje jeden či více kontejnerů. Při spuštění je do kontejneru registrována uživatelova aplikace. (Tento kontejner tedy nemá nic společného s terminologií používanou ve virtualizaci na úrovní operačního systému v Linuxu. Jedná se čistě o serverovou aplikaci.) Zároveň zde kontejner nevystupuje jako konkrétní aplikace, nýbrž jako abstraktní software, který má svůj životní cyklus a může mít závislosti na jiném software.

Nasazení aplikace do reálného prostředí, které je velmi blízké tomu produkčnímu, je velmi důležité. V opačném případě se jedná o aproximaci reálného, produkčního prostředí a některé chyby se tak nemusí projevit.

Projekt Arquillian automatizuje správu kontejneru, ve kterém je aplikace nasazena, nasazení aplikace do kontejneru a inicializaci použitého frameworku. Programátor se tak nemusí zabývat režií procesů předcházejících spuštění testu a může se soustředit především na psaní testů. Arquillian tedy přináší zejména automatizaci a pro samotné testování v jednotlivých fázích jsou využity dostupné frameworky. Kromě JUnit lze využít konkurenční testovací framework TestNG a pro funkční nebo akceptační testování je možno psát scénáře pro Selenium. Arquillian tyto frameworky doplňuje o[8]:

- Správa životního cyklu kontejneru, ve kterém je aplikace testována.
- Zabalení testů a potřebných závislostí do archívu *ShrinkWrap* (ten bude představen dále v této kapitole).
- Nasazení archívu v kontejneru.
- Podpora slabých vazeb mezi objekty pomocí dependency injection.
- Spouštění testů uvnitř kontejneru.
- Kolekce výsledků testů a následná validace v rámci frameworku použitého pro testování (JUnit...).

Cílem tohoto projektu je zvýšení kvality softwarového produktu jako celku skrze časté testování s důrazem na automatizaci procesů, jako je překlad zdrojových souborů a nahrávání aplikace na server. Autoři projektu tento přístup nazývají *Continuous Development*.

V rámci tohoto přístupu definují několik pravidel, která mají za cíl zkvalitnit nejen vývoj produktu, ale i následný proces nasazení do provozu a proces údržby. Jedno z těchto pravidel je důraz na přenositelnost produktu mezi všemi kontejnery, které splňují patřičnou specifikaci. Jak bylo uvedeno v kapitole 2, pro daný standard (např. Java EE) často existuje kromě referenční implementace i jedna či více dalších implementací. Ty často přidávají proprietární funkcionalitu, jejímž použitím se projekt k této implementaci váže. Nepoužívání nestandardní funkcionality podporuje přenositelnost aplikace mezi jednotlivými kontejnery, což při dlouhém životním cyklu aplikace znamená flexibilitu ve chvíli, kdy je z nějakého důvodu potřeba přejít na jiný kontejner. Další základní princip je zajištění rychlého spuštění celé testovací sady. Testu tedy nesmí předcházet složitá manuální příprava, test má být ideálně spuštěn stiskem tlačítka v IDE[10] nebo napsáním jednoho příkazu v příkazové řádce.

ShrinkWrap

Častý překlad zdrojových kódů, spuštění a zastavení kontejneru, jsou operace intenzivně využívající souborový systém, což je v dnešních počítačích úzké hrdlo výkonu. Mnohem větší nevýhoda je však v případě, kdy test není spouštěn v nějakém lokálním kontejneru, určeném pouze pro testování, ale v kontejneru poskytovaném aplikačním serverem, který běží dlouhodobě na nějakém vzdáleném počítači. Typicky někde v serverovně. Při každém spuštění testu jsou totiž do souborového systému tohoto počítače zapsána a po skončení testu odstraněna data. To je odlišný princip od produkčního stroje, kde je aplikace jednou nainstalována a poté jsou inkrementálně přidávány aktualizace. Jak již bylo řečeno, v rámci *Continuous Development* je snaha testovat aplikace v prostředí, které je totožně s produkčním prostředím, ve kterém dokončená aplikace má být nasazena. Tento problém nevzniká pouze při testování a tak Arquillian využívá již osvědčené řešení – virtualizaci souborového systému.

ShrinkWrap je virtuální souborový systém. Nejedná se o přímou závislost projektu Arquillian, i když v této komunitě vývojářů vznikl. Arquillian je možné použít bez *ShrinkWrap* stejně jako *ShrinkWrap* bez Arquillianu. V předchozím odstavci byla popsána snaha automatizovat a minimalizovat čas potřebný pro spuštění testovací sady, to zajistí projekt Arquillian a dodržování principů, které definuje. Samotný zdrojový kód je nicméně potřeba přeložit a poté sestavit výslednou aplikaci (*build*) se všemi závislými objekty i zdroji mimo zdrojové kódy. Tento *build* je poté zapsán na disk v komprimované podobě. Pro spuštění aplikace v kontejneru je pak třeba data z disku načíst. To je krok navíc, který u rozsáhlejších projektů netrvá sekundy, ale desítky sekund. Proto je využit virtuální souborový systém a přeložená aplikace společně se svými závislostmi je uchována pouze v paměti. Jednoduchý příklad *ShrinkWrap* archívu:

```
JavaArchive archive = ShrinkWrap.create(JavaArchive.class,"myarchive.jar")
   .addClasses(MyClass.class, MyOtherClass.class)
   .addResource("mystuff.properties");
```

Příklad 2.9: *Archív ShrinkWrap*

[10] Integrated Development Enviroment

Arquillian a JUnit

Na konkrétním příkladu z oficiální dokumentace[1] bude popsáno testování Java aplikace pomocí JUnit. Pro vytvoření archívu s byte kódem a potřebnými závislostmi bude použit *ShrinkWrap*. Jako testovaná aplikace bude použita třída `Greeter`, představená v předchozí kapitole. Bude se tedy jednat stále o unit test, nicméně tentokrát nebude aplikace spuštěna samostatnou JVM, ale bude nasazena do kontejneru Arquillian Weld[11]. Pro připomenutí následuje zdrojový kód:

```
public class Greeter {
  public String createGreeting(String name) {
    return "Hello, " + name + "!";
  }
}
```

Příklad 2.10: *Testovaná třída Greeter*

Jelikož testovací případ pro tuto aplikaci obsahuje anotace z mnoha frameworků, jsou uvedeny i řádky s importovanými závislostmi, aby bylo možné rozeznat, co která anotace znamená.

```
import javax.inject.Inject;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.Test;
import org.junit.Assert;
import org.junit.runner.RunWith;

@RunWith(Arquillian.class)
public class GreeterTest {

  @Deployment
  public static JavaArchive createDeployment() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClass(Greeter.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  Greeter greeter;

  @Test
  public void testCreateGreetingInContainer() {
    String expected = "Hello, Earthling!";
    String result = greeter.createGreeting("Earthling");
    Assert.assertEquals(expected, result);
  }
}
```

Příklad 2.11: *Arquillian test třídy Greeter*

Použití anotací je velmi zajímavý model, jednoduchými výrazy jsou metody a proměnné „označeny“ a dle těchto značek jsou pak zpracovány logikou skrytou uvnitř frameworku.

[11] Více informací zde: `http://arquillian.org/modules/arquillian-weld-ee-embedded-1.1-container-adapter/`

Při čtení zdrojového kódu je důležité konfrontovat danou anotaci s knihovnou, ze které je importována. První použitá anotace `@RunWith` je součástí JUnit a umožňuje použít jinou třídu, která převezme řízení testování namísto od defaultní třídy JUnit. V tomto případě se o kontrolu testu stará Arquillian, je totiž potřeba nalézt a zpracovat anotace `@Deployment` a `@Inject`, které nejsou součástí JUnit. `@Deployment` identifikuje metodu, která vytváří virtuální archív pomocí *ShrinWrap*. V tomto příkladu je součástí archívu i prázdny soubor `beans.xml`, přítomnost tohoto souboru v archívu je totiž nutná pro aktivaci dependency injection – koncept správy závislostí ve smyslu slabé vazby mezi objekty, které nejsou ve vztahu objekt vlastní jiný objekt (a tak řídí jeho životní cyklus), ale ve vztahu objekt ví o jiném objektu. V tomto případě je anotace `@Inject` použita pro objekt `Greeter`, třída tohoto objektu je vyhledána v globální proměnné `CLASSPATH`, Arquillian vytvoří instanci tohoto objektu a ta je vložena (*inject*) do proměnné `greeter`. Flexibilita tohoto principu leží v tom, že ve zdrojovém kódu není určeno, který objekt bude použit, ale až za běhu je vybrán objekt dle potřeb, přičemž to může být jakýkoli objekt odvozený od `Greeter`.

2.4 Testování pomocí Pax Exam

Pax Exam je framework pro automatizované testování aplikací běžících uvnitř OSGi kontejneru. V této kapitole bude představeno OSGi a způsob, jakým Pax Exam ulehčuje testování aplikací běžících v takovém kontejneru s použitím frameworku JUnit.

OSGi

Hranice mezi částmi aplikace na objektové úrovni jsou součástí „dobrého návrhu“, nikoli standardu. Aplikace založená na modulárním návrhu je složená z modulů, které mají definované role a rozhraní, přes které komunikují s okolními moduly. Modularita tedy není vynucená a i když je během návrhu kladen důraz na udržení konzistence jednotlivých modulů (Erich Gamma: „Programujte oproti rozhraní, ne implementaci.“[12][4]), není nijak zajištěno, že výsledný systém bude modulární. To se snaží změnit komunita OSGi Alliance specifikací *Open Service Gateway initiative*. Popisuje kontejner, který vynucuje modularitu. Základní charakteristiky tohoto kontejneru jsou isolace jednotlivých modulů, identifikace jejich závislostí a poskytování služeb těmto modulům. Oproti integraci software se zde jedná o subsystémy v rámci jedné aplikace, nikoli o komunikaci více aplikací.

OSGi je koncept přidávání abstrakce, s růstem aplikace (přidáváním tříd) roste komplexita a i když třídy jsou postaveny na základech enkapsulace a rozhraní, při velkém počtu tříd a především velkém počtu vztahů mezi třídami (závislostí) je tato základní modularita příliš jemné dělení. Jeden modul OSGi tak zpravidla zapouzdřuje více tříd a vytváří tím logický celek s jednotnou funkcionalitou. Tyto třídy jsou dle konvencí Javy seskupovány do balíků a ty zase do *.jar* archívů. OSGi modul je pak kolekce Java archívů. Jednomu modulu se říká *bundle* a v rámci jednoho modulu jsou všechny balíky navzájem přístupné. Zvenku modulu jsou pak přístupné pouze ty balíky, které jsou explicitně exportovány pro použití ostatními moduly. Ilustračním příkladem může být modul, který obsahuje pouze jediný balík – rozhraní, další modul pak obsahuje balík s implementací tohoto rozhraní. Implementace tohoto rozhraní je skryta a exportováno je pouze rozhraní, přes které mohou ostatní balíky tuto implementaci používat. Tím je zajištěna modularita v rámci aplikace i při velkém počtu tříd a co je nejdůležitější, tato modularita je zajištěna i za běhu aplikace pomocí OSGi kontejneru.

[12] program to interfaces, not to implementations

Dobrým příkladem použití OSGi je vývojové prostředí Eclipse vyvíjené open source komunitou Eclipse Foundation. Tato komunita vytvořila vlastní implementaci kontejneru dle specifikace OSGi (*Equinox*) a po nainstalování samotného IDE je možné přidávat a odebírat pluginy (např. PyDev pro přidání zvýrazňování syntaxe a dalších funkcí pro jazyk Python), které jsou implementovány jako OSGi *bundle*.

Pax Exam a JUnit

OSGi je běhové prostředí, před tím, než může hostovat nějaké aplikace, je potřeba jej spustit a inicializovat. Manuální spuštění znamená spuštění procesu z terminálu, správa životního cyklu aplikace je pak dostupná pomocí konzole konkrétní implementace specifikace OSGi, většina implementací nabízí i webové rozhraní. Spuštění testu v kontejneru tradičně znamená nasazení testované aplikace společně s testovacím případem a po jeho skončení je třeba získat výsledek. Automatizace tohoto procesu je zprostředkována právě projektem Pax Exam. Pro konfiguraci kontejneru tento framework definuje anotaci `@Configuration`, kterou musí být v každé testovací sadě anotována právě jedna metoda, jejíž návratovou hodnotou je pole objektů `Option`. Zde programátor musí specifikovat, kterou implementaci OSGi chce použít. Součástí Pax Exam jsou funkce pro konfiguraci konkrétních implementací, například `felix()` pro běh aplikace v OSGi implementaci Apache Felix. Dále je potřeba zde specifikovat moduly, na kterých testovaná aplikace závisí a je tak nutné je zahrnout do výstupní aplikace. Dále je zde možno upravit konfigurace testu, například četnost logovacích zpráv atd. Viz dokumentace[12].

Dále je potřeba anotací `@RunWith` změnit výchozí třídu řídící testování na třídu poskytovanou frameworkem Pax Exam: `JUnit4TestRunner`. Ta zajišťuje zpracování anotací mimo JUnit a režii pro správu kontejneru. Následuje jednoduchý příklad testu Pax Exam:

```
@RunWith(JUnit4TestRunner.class)
public class GreeterInOSGiTest {

  @Inject
  private Greeter greeter;

  @Configuration
  public Option[] config() {
    return options(
        felix(),
        junitBundles()
      );
  }

  @Test
  public testCreateGreetingInOSGi() {
    String expected = "Hello, Pax!";
    String result = greeter.createGreeting("Pax");
    assertEquals(expected, result);
  }
}
```

Příklad 2.12: *Pax Exam test třídy Greeter*

Jak je vidět, Pax Exam i Arquillian přidávají zejména podporu pro automatizovanou inicializaci specifických frameworků. Oba vychází z Java EE a JUnit a tak jsou ve způsobu zápisu testů v těchto frameworcích společné rysy. Nahrazení *test runner* třídy třídou podporující funkce frameworku, konfigurace, dependency injection a samotné testovací případy.

Tento příklad testuje čtenáři již dobře známou aplikaci `Greeter`, nyní samozřejmě v OSGi kontejneru. Ten je konfigurován voláním metody `options`. První argument specifikuje konkrétní OSGi kontejner, ve kterém aplikace bude spuštěna a druhý argument je funkce frameworku identifikující závislosti pro framework JUnit. Tyto zkompilované archívy budou slinkovány a nasazeny do OSGi kontejneru a po skončení testu budou výsledky navráceny zpět do JUnit, kde dojde k jejich vyhodnocení.

2.5 Testování pomocí Apache Camel

Projekt Apache Camel byl založen pro usnadnění integrace software, což je poměrně komplikovaná oblast věnující se budování rozsáhlých, distribuovaných systémů. Před vytvořením knihoven jako je Camel vývojářské týmy řešily jak propojit distribuované komponenty a bylo potřeba mít v týmu dedikovaného člověka s potřebnou specializací. Ani to však nezabránilo neustálému znovu-vynalézání kola, jelikož potřebné znalosti měl jen malý počet lidí. Projekt Camel je v adresování tohoto problému konkrétní implementací, dostupná jako open source, s komunitou odborníků na tuto oblast.

Běžný nárok na takovéto systémy může být spolehlivé zasílání zpráv, což se na první pohled nemusí zdát jako příliš náročný požadavek. V systému s mnoha uzly se však vyskytují zcela nové problémy, například rychlost šíření zprávy v systému – bude zpráva doručena všem uzlům včas[13]? A co se stane, pokud nastane nějaká chyba, například jeden či více uzlů přestane fungovat?

Následuje krátký úvod do integrace software, poté jsou vysvětleny základy, na kterých je Apache Camel postaven. Poslední část se věnuje pouze projektu Camel a testování Camel aplikací s využitím JUnit.

Integrace software

Integrace počítačových aplikací znamená vytváření systému složeného z n podsystémů. Typickým podsystémem při integraci software je proces hostovaný operačním systémem. Z toho vyplývá, že integrace je založena na meziprocesové komunikaci a integrovaný systém je pak n-tice komunikujících procesů. Jednoduchý příklad mohou být dva procesy operačního systému Linux komunikující přes sdílenou paměť – lokální meziprocesová komunikace. Pro vzdálenou meziprocesovou komunikaci je nejčastěji použita komunikace nad architekturou TCP/IP, jejíž implementace je programátorovi přístupná přes rozhraní schránek (sockets).

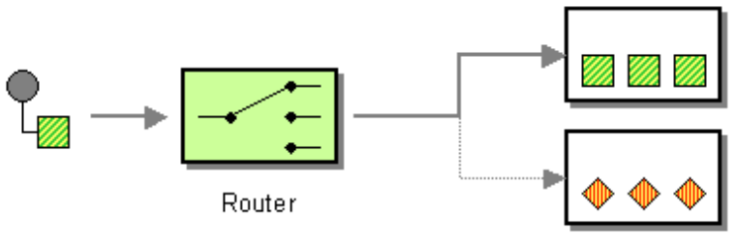
Dnes jsou často používány knihovny vnášející abstrakci nad čistě systémová volání, na kterých jsou schránky založeny. Těchto knihoven existuje mnoho a tak existuje i mnoho přístupů k řešení a také různých abstrakcí. Například knihovna ZeroMQ přináší abstrakci nad samotnou síťovou komunikací. Jiné knihovny přidávají asynchronní komunikaci, spolehlivé doručování zpráv, zotavení se z chyb a některé i zabezpečení přenosu.

Enterprise Integration Patterns

To co kniha *Design Patterns: Elements of Reusable Object-Oriented Software*[4] znamená pro návrh objektově orientovaného software kniha *Enterprise Integration Patterns*[6] znamená pro integraci software. Obsahuje 65 často se vyskytujících problémů v oblasti integrace software a jejich řešení.

[13] Douglas Schmidt: The right answer delivered too late becomes the wrong answer.

Příklad vzoru je *content based router*. Systém složen z jedné aplikace generující zprávy a n aplikací konzumujících tyto zprávy s tím, že jedna zpráva má právě jednoho příjemce. Mezi generující aplikací a konzumujícími aplikacemi je umístěn *content based router*, který zprávy na základě jejich obsahu klasifikuje a přeposílá k příslušným konzumentům.



Obrázek 2.1: *Content based router*

Tato kniha byla vydána v roce 2003 a od té doby vzniklo několik frameworků, které uvedené vzory implementují. Například Apache Camel, na který se tato práce zaměřuje. Další implementace je Spring Integration.

Motivace pro použití Apache Camel

Zmíněné vzory jsou návody, jak zpracovat zprávu. Ta ale musí nejprve přijít od odesílatele, který komunikuje určitým protokolem a samotná zpráva je reprezentována určitým formátem. Výraznou vlastností projektu Camel je způsob, jakým umožňuje jednoduše přidávat podporu pro komunikační protokoly a formáty dat. Kromě textových formátů jako XML, JSON, CSV a dalších umí překládat data z a do formátů binárních, včetně komprese a šifrování. V současné době je součástí Camelu přes 150 konvertorů mezi formáty dat (včetně barkódu i jiných méně častých reprezentací dat). Pokud je použit proprietární formát, lze si jeho podporu naprogramovat. V závislosti na aplikacích, které jsou propojovány je automaticky provedena konverze formátu tak, aby přijímající strana vždy zprávě rozuměla.

Po přijetí dat následuje fáze zpracování těchto dat v závislosti na požadavcích dané úlohy. V rámci *Enterprise Integration Patterns* existuje mnoho předem připravených řešení. Tím se vývojáři vyhýbají znovu-vynalézání kola a nové algoritmy programují pouze ve chvíli, kdy tyto vzory jejich potřeby nesplňují.

Další oceňovanou vlastností je, že uvedená funkcionalita není limitována pouze na jazyk Java. Lze ji použít i ve Scale, Groovy a pomocí XML i ve frameworku Spring. Vlastní procedury v XML samozřejmě psát nejde, ale jde se odkázat na Java kód (Spring je napsán v Javě). Obecný, ale zároveň velice vystihující popis Apache Camel lze vyjádřit takto:

- Přijmutí dat z jakéhokoli zdroje v jakémkoli formátu.
- Zpracování těchto dat standardními metodami nebo vlastními.
- Odeslání dat jakémukoli cíli v jakémkoli formátu.

Výhodou použití projektu Camel oproti programování vlastního řešení je dostupná podpora široké škály formátů, důraz na dodržování standardů a směrování dat v systému podle EIP. Všechny výše popsané operace se zapisují pomocí doménově specifického jazyka, který Camel implementuje v rámci jazyků Java, Scala a Groovy. Přidávání vlastních funkcí je založeno na vytváření objektů dle konvence *POJO*[14], (někdy nazývána *Beans*) – objekt není

[14] Plain Old Java Object

spjat s žádným frameworkem ani knihovnou, obsahuje konstruktor bez argumentů, všechny instanční proměnné jsou dostupné metodami *getters* a *setters* a je serializovatelný. Tato konvence podporuje slabé vazby mezi objekty v objektově orientovaném návrhu a je považována za dorbý základ pro udržování a testování rozsáhlých aplikací.

Úvod k Apache Camel

Základními prvky jsou koncové body (*endpoints*), ty se aktivně podílejí na komunikaci – od odesílatele konzumují zprávy a adresátovi je zase produkují. Zprávy jsou dvou druhů:

- Zpráva (*message*) – jednosměrná zpráva, obsahuje hlavičku a tělo zprávy, rezervovaný prostor pro možnou přílohu.
- Výměna (*exchange*) – zapouzdřuje jeden pár zpráv, odchozí a příchozí. Obsahuje prostor pro objekt výjimky v případě chyby, identifikátor výměny, specifikátor určující přítomnost pouze odchozí zprávy nebo i příchozí. Prostor *properties* je použit jako hlavička celé výměny.

Mezi koncovými body je cesta, na níž se mohou nacházet prvky zpracovávající a/nebo směrující zprávy mezi jednotlivými body. Tyto prvky jsou označovány jako procesory (*processor*) – zde se nacházejí EIP. Příkladem je již zmiňovaný *content based router*.

Camel je dělen na komponenty, základní komponentou je *camel-core*, která obsahuje nejesenciálnější prvek Camelu – *CamelContext*. Všechny ostatní komponenty jsou pak přidávány a odebírány podle potřeb. Pro konkrétní aplikaci tedy není nutné překládat a nahrávat na server všechny komponenty, ale pouze ty, které jsou danou aplikací využity. Použité komponenty jsou pak zapouzdřeny v *CamelContext*. Každý program využívající Apache Camel musí vytvořit objekt třídy *CamelContext*, tím je provedena základní inicializace, dále je nutné zaregistrovat použité komponenty, koncové body, cesty, procesory a ostatní prvky Camelu, které budou v aplikaci použity. Obsah *CamelContext* je znázorněn na obrázku 2.2.

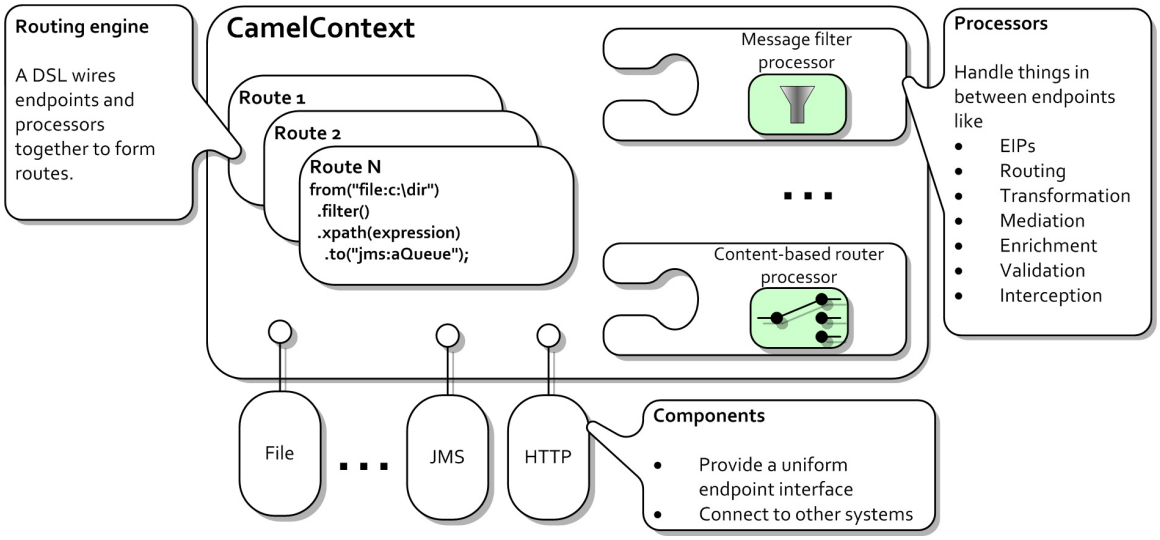
Příklad velmi jednoduché aplikace z knihy *Camel in Action*[7]:

```
public class FileCopierWithCamel {
  public static void main(String args[]) throws Exception {
    CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
      public void configure() {
        from("file:data/inbox?noop=true").to("file:data/outbox");
      }
    });

    context.start();
    Thread.sleep(10000);
    context.stop();
  }
}
```

Příklad 2.13: *Hello World příklad v Apache Camel*

Camel následuje praktiku *Convention over Configuration* – poskytnutí defaultní konfigurace tak, že ve většině případů je možné ihned začít knihovnu používat a jen ve velmi specifických případech je nutné provést zásah do konfigurace. Proto je knihovna v uvedeném příkladu inicializována instanciací objektu `DefaultCamelContext`. Poté je invokováním metody `addRoutes` knihovně předán objekt `RouteBuilder` (v uvedeném příkladu vytvořen

jako anonymní objekt), který obsahuje jedinou metodu. V metodě `configure` je pomocí doménově specifického jazyka, který Camel pro definici cest používá, vytvořena nová cesta. V tomto případě velice jednoduchá, neobsahuje žádný procesor, pouze dva koncové body – – bod konzumující z adresáře *inbox* v adresáři *data* a bod produkující do adresáře *outbox* v témže adresáři. Parametrem *noop* je specifikováno, že po provedení operace nemá být soubor z adresáře inbox smazán. Poté je aplikace spuštěna a následně zastavena voláním metod kontextu. Hlavní vlákno je uspáno na konstantní dobu pro zajištění, že operační systém stihne obsloužit požadavky pracující se souborovým systémem.



Obrázek 2.2: *Architektura projektu Apache Camel*[7]

Testování aplikací Apache Camel

Významnou roli v tomto druhu testování hraje abstrakce nad reálnou komponentou systému, kdy je reálná komponenta nahrazena zjednodušením pouze na množinu vlastností, které jsou potřeba v daném testovacím případě. Takovéto abstrakce lze zanášet v rámci komponent, které komunikují s testovanou komponentou. Samotnou testovanou komponentu samozřejmě nahradit nelze. Vytváření testovacích případů je tedy do jisté míry modelováním systému a vytvářením patřičných abstrakcí. Pro vytváření abstraktních komponent je součástí *camel-core* třida `MockEndpoint`. Zde je seznam situací, kdy je vhodné zavádět abstrakce nad reálnými komponentami[7]:

- Reálná komponenta není ve fázi testování testované komponenty dostupná.
- Použití reálné komponenty je příliš pomalé nebo potřebuje náročnou inicializaci (databáze...).
- Do reálné komponenty je třeba přidat další funkcionalitu, bez které testování není možné.
- Reálná komponenta vrací nedeterministická data, např. závislá na čase.
- Dané scénario zahrnuje simulaci chyb, například chyba v dané komponentě nebo chyba sítě.

Další součástí *camel-core* užitečnou při testování je `ProducerTemplate`, třída která umožňuje vytvářet vlastní zprávy, které jsou pak zaslány do systému. Testování Camel aplikací je založeno na frameworku JUnit, který *camel-test* rozšiřuje o další funkcionalitu. Jsou tedy dostupné všechny vlastnosti JUnit a navíc vlastnosti specifické pro Camel a několik vlastností navíc, které zvyšují komfort programátora při psaní testovacích případů, například jednoduchá práce se souborovým systémem nebo inicializace testovacího prostředí. Třída zapouzdřující tuto funkcionalitu pro JUnit ve verzi 4 se jmenuje *CamelTestSupport*.

Principem testování Camel aplikací je zaslání zprávy do systému a následné ověřování výstupu systému. Příklad velmi jednoduchého testovacího případu z knihy *Camel in Action*[7]:

```
public class FirstTest extends CamelTestSupport {
  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
    //Vytvoření cesty a koncových bodů, které se budou testovat.
    return new RouteBuilder() {
      @Override
      public void configure() throws Exception {
        from("file://target/inbox").to("file://target/outbox");
      }
    };
  }

  @Test
  public void testMoveFile() throws Exception {
    //Pomocí ProducerTemplate odeslat zprávu do koncového bodu inbox.
    template.sendBodyAndHeader("file://target/inbox", "Hello World",
        Exchange.FILE_NAME, "hello.txt");

    //Ověření existence souboru.
    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    //Ověření obsahu souboru.
    String content = context.getTypeConverter().convertTo(String.class, target);
    assertEquals("Hello World", content);
  }
}
```

Příklad 2.14: *Test Hello World příkladu*

Jak je vidět na uvedeném příkladu, testování Camel aplikací kombinuje vlastnosti JUnit a Camelu. Data v systému jsou generována Camelem, ověřování prováděno metodami JUnit.

V další ukázce bude představeno použití *Mock* komponenty namísto reálné komponenty. JUnit tento způsob testování nepodporuje, proto je součástí třídy `MockEndpoint` metoda `assertIsSatisfied`, která testuje, zda test neselhal. Na začátku každého testovacího případu jsou nastavena očekávání (*expectations*) pro daný případ, poté je test spuštěn a po jeho dokončení je provedena kontrola, zda všechna očekávání byla úspěšně naplněna. Očekávání mohou být:

- Počet doručených zpráv pro daný koncový bod.
- Minimální počet doručených zpráv pro daný koncový bod.
- Počet těl ve zprávě. Záleží na pořadí, v jakém dorazí na daný *endpoint*.
- Počet těl ve zprávě. Nezáleží na pořadí.

Příklad použití *Mock* namísto reálné komponenty z knihy *Camel in Action*[7]:

```
public class FirstMockTest extends CamelTestSupport {
  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
      @Override
      public void configure() throws Exception {
        from("jms:topic:quote").to("mock:quote");
      }
    };
  }

  @Test
  public void testQuote() throws Exception {
    //Získání reference na cílový endpoint
    MockEndpoint quote = getMockEndpoint("mock:quote");
    //Nastavení očekávání pro daný testovací případ
    quote.expectedMessageCount(1);
    //Odeslání zprávy
    template.sendBody("jms:topic:quote", "Camel rocks");
    //Ověření naplnění očekávání
    quote.assertIsSatisfied();
  }
}
```

Příklad 2.15: *Test s použitím MockEndpoint*

V tomto testu je jako konzumující bod použito rozhraní Java Messaging Services s názvem *quote* a produkující bod je *Mock* objekt, se kterým je asociováno očekávání přijetí právě jedné zprávy. Na konci je proveden test naplnění očekávání, v tomto případě je očekávání pouze jedno a je demonstrativně jednoduché. V reálném testovacím případě se postupuje dle stejného principu, očekávání jsou však kombinována pro pokrytí testovacích kritérií daného scénáře.

2.5.1 Vlastní přínos pro projekt Apache Camel

Cílem práce je vybrat vhodné metody a nahradit je anotacemi. Nastává tak problém nalézt testovací třídu s anotacemi, což při prohledávání velkého počtu tříd může trvat nepřijatelně dlouho. Proto jsem se po konzultaci s komunitou projektu rozhodl napsat nový *test runner* pro Camel, což problém prohledávání všech tříd v proměnné `CLASSPATH`[15] eliminuje – třída obsahující testy je *test runner* třídě známá již při spuštění. Kromě nalezení a zpracování anotací je tak potřeba refaktorovat stávající konfiguraci kontextu a všech funkcionalit určených pro testování tak, aby vyhovovaly nové *test runner* třídě. S tímto rozhodnutím je spojen další přínos a to odstranění přímých závislostí testovací třídy na interních mechanismech Camelu, které vznikají dědičností. V mém řešení má testovací třída přístup pouze k objektům, které deklaruje a k objektům, které jsou do testovací třídy vloženy *test runner* třídou pomocí dependency injection.

[15] Proměnná uchovávající cesty k .class souborům.

Kapitola 3

Návrh a implementace anotací pro testovací komponentu Apache Camel

V předchozí kapitole byl představen projekt Camel a poslední část se věnovala jeho komponentě *camel-test*, která přidává podporu pro testování Camel aplikací, tedy přípravu *CamelContext* a přidání několika funkcnionalit pro ulehčení psaní testů. Jelikož současná implementace *camel-test* je založena na JUnit, bude následovat analýza vnitřních mechanizmů JUnit a také budou nastíněny různé způsoby, jakými lze docílit požadovaného výsledku – – nahrazení některých metod anotacemi. Poté budou popsány anotace, které nahrazují vybrané metody z balíku *camel-test* a popis nového API.

3.1 Hledání anotací

Jelikož tato práce nezačíná na zelené louce, ale vychází z již existujících projektů, možnosti jak vyhledávat anotace jsou určeny nejen standardními prostředky, ale i návrhem zmíněných projektů. V případě JUnit ve velice pozitivním smyslu – výpočetně nejnáročnější je nalézt třídu, ve které se nachází anotace (v tomto případě třídu s testy) a ta je JUnit vždy známá, je totiž předávána při spuštění jako argument.

Kdyby tato třída nebyla známá před spuštěním programu, bylo by nutné prohledávat všechny třídy nebo se nějakým způsobem snažit redukovat počet tříd, které je nutné při hledání navštívit. Například podle nějaké jmenné konvence. Jelikož v JUnit jsou často názvy testovacích tříd zakončeny postfixem „Test“, naskýtá se řešení prohledávat pouze ty třídy, jejichž název vyhovuje regulárnímu výrazu „`.*Test`“. Což vytváří limit možných platných názvů tříd a tak tato možnost byla zavrhnuta.

V kapitole o JUnit byla popsána role *test runner* a zde je již jasné, že pro přidání nových funkcí je potřebné napsat *test runner* pro Camel, který bude zapouzdřovat jak vyhledání a zpracování anotací, tak všechnu funkcionalitu dostupnou v *camel-test*. Tato komponenta totiž byla napsána před vydáním JUnit verze 4.6, od které je možné modifikovat JUnit pomocí *test runner* tříd a tak je napsána dnes již zastaralým způsobem. V této kapitole je část věnující se současnému návrhu této komponenty a budou tak probrána úskalí plynoucí z její implementace.

3.2 Analýza JUnit

V kapitole o JUnit byla popsána stromová struktura plánovače událostí, které je nutné provést během testu. Nyní bude popsán algoritmus, kterým plánovač události vybírá a poté modifikace algoritmu pro účely Camel. Konkrétně je nutné přidat inicializaci *CamelContext* a všech procedur s tím spojených a také úklid po dokončení testování.

Událost v JUnit je třída dědící od abstraktní třídy `Statement`, která deklaruje jedinou metodu: `void evaluate`, ve které může být volána stejná funkce jiného `Statement` objektu. Precedence je poté určena dle pořadí operací v těle metody `evaluate` – první operace je buď volání metody `evaluate` jiného objektu, což znamená, že volaná metoda je součástí objektu s vyšší prioritou. Volaný objekt má nižší prioritu v situaci kdy je napřed proveden algoritmus vztahující se k vlastnímu objektu a až poté je volána metoda jiného `Statement` objektu. Algoritmus končí ve chvíli, kdy není volána další metoda `evaluate`. Lepší představu lze získat z příkladu 3.1, kde uvedená událost má vyšší prioritu než událost předávaná konstruktoru.

```
public class ExampleStatement extends Statement {

  private Statement lowerPriority;

  public ExampleStatement(Statement lowerPriority) {
    this.lowerPriority = lowerPriority;
  }

  @Override
  public void evaluate() {
    runLocalAlgorithm();      // Provedení algoritmu vlastního objektu
    lowerPriority.evaluate(); // Provedení algoritmu jiného objektu
  }
}
```

Příklad 3.1: *Příklad Statement objektu JUnit*

Může se jednat o události spjaté s akcemi na úrovni testovací metody, tedy události bezprostředně přechzázející a následující volání metody anotované `@Test`. V takovém případě se jedná o blok událostí dějící se na úrovni testovací metody. Naopak metody anotované `@BeforeClass` a `@AfterClass`, jak bylo již dříve uvedeno, se k testovacím metodám nevztahují, vztahují se pouze k testovací třídě a každá je provedena nanejvýš jednou. Proto jsou označovány jako třídní blok. V příloze A.1 je uveden diagram tříd, na kterém je možné sledovat hierarchii zpracování událostí – metoda `classBlock` v abstraktní třídě `ParentRunner` implementuje zpracování právě metod třídního bloku, zároveň se zde nachází implementace všech operací na úrovni testovací třídy. Třída `BlockJUnit4ClassRunner` je defaultní *test runner* JUnit a implementuje operace na úrovní testovací metody. Jelikož požadavkem pro rozšíření testovací komponenty Apache Camel je pouze přidat do JUnit operace navíc, spjaté s inicializací frameworku, je třída `BlockJUnit4ClassRunner` vhodným výchozím bodem. Děděním od této třídy budou zachovány všechny operace výchozí *test runner* třídy a redefinicí minimálního počtu metod se dá docílit rozšíření projektu Camel, které je plně v souladu se současnými principy v JUnit.

3.3 Analýza testovací komponenty Apache Camel

Aby bylo možné Camel aplikace jednoduše testovat, vznikla k tomu určená komponenta založená na JUnit 4.0. Aktuální verze JUnit v době vzniku této práce je 4.12 a přepra-

cování testovací komponenty tak, aby odpovídala aktuální verzi JUnit je pochopitelným požadavkem. Současná implementace má několik nechtěných vlastností, tou zásadní je, že operace zajišťující inicializaci *CamelContext* jsou součástí testu, i když nepřímo přes dědičnost. Jak je možné vidět na ukázkách testů v sekci 2.5, každý testovací případ dědí od třídy `CamelTestSupport`, kde se nachází právě inicializace. Je zde využito toho, že JUnit hledá anotace v celé třídní hierarchii testovacího případu. Volání anotovaných metod je poté ve stejném pořadí, jako je volání konstruktorů, tedy shora dolů. Díky tomu se ve třídě `CamelTestSupport` může vyskytovat metoda anotovaná `@Before` pro inicializaci kontextu a metody anotované `@After` a `@AfterClass` pro úklid. `CamelTestSupport` dále dětí od třídy `TestSupport`, kde se nachází funkce pro usnadnění testování, například odstranění adresáře atd. Tato třída dědí od třídy `Assert`, což je jedna ze základních tříd JUnit, nachází se v ní všechny *assert* metody. To má za následek, že každý Camel test má zpřístupněné všechny *assert* metody, bez ohledu na to, jestli jsou použity nebo ne. Tuto silnou vazbu testovací třídy na třídu `Assert` a třídy Camel lze odstranit přidáním nové *test runner* třídy, která bude obstarávat operace potřebné pro zprostředkování funkcí Camel v rámci testovací třídy a to právě zavedením nových anotací, specifických pro Apache Camel. V příloze A.2 je diagram tříd znázorňující implementaci *camel-test* pro JUnit verze 4.0.

3.4 Návrh anotací

Pro shrnutí následuje výčet vlastností, které je nutné odstranit:

- Závislost testu na třídě `Assert` – bude nahrazeno importy pouze těch metod, které jsou v testu použity.
- Závislost testu na třídě `TestSupport` – stejně jako v předchozím bodě, metody lze samostatně importovat.
- Závislost testu na třídě `CamelTestSupport` – bude vytvořena nová *test runner* třída, která pomocí dependency injection zprostředkuje vnější závislosti testovací třídy.

Nová *test runner* třída bude pojmenována `CamelRunner` a zbytek této práce se bude na tento název odkazovat. Jak bylo naznačeno dříve, je vhodné tuto třídu zapojit do existující hierarchie řízení testování a pouze přidat operace navíc spojené s projektem Camel, tedy `CamelRunner` dědí od třídy `BlockJUnit4ClassRunner`, ze které redefinuje metodu `Statement methodBlock`, do které jsou přidány objekty `Statement` pro inicializaci *CamelContext* a úklid. Ze třídy `ParentRunner` je redefinována metoda `Statemnet classBlock`, do které je přidán objekt `Statement` pro úklid na konci testování – během testu bylo spuštěno více vláken, které je nutné korektně ukončit. Anotace:

- `@ContextInject` – tato anotace slouží pro zpřístupnění *CamelContext* v rámci testu. Lze s ní anotovat třídní proměnné typu `CamelContext`, které musí mít veřejný modifikátor přístupu.

Nyní je v rámci testu možné používat *CamelContext*, ke kterému se registrují cesty definující koncové body a pravidla pro směrování. V první ukázce v kapitole o Apache Camel byla cesta přidána v rámci uživatelem definované třídy. V ukázkách testovací komponenty uživatel již cesty pouze definoval a přidány byly právě testovací komponentou. Toho bylo dosáhnuto tak, že uživatel redefinoval metodu `RouteBuilder createRouteBuilder`. S `RouteBuilder`

je spojena ještě jedna metoda: `boolean isUseRouteBuilder`. Její defaultní implementace v rámci třídy `CamelTestSupport` vrací hodnotu *true*, tedy registrace cest do kontextu je povolena, pomocí této metody ji lze také zakázat. Tyto dvě metody jsou nahrazeny následujícími anotacemi:

- `@UseRouteBuilder` – lze anotovat pouze třídu, jeden formální argument typu `boolean`, jehož defaultní hodnota je *true*.
- `@CreateRouteBuilder` – slouží k vytvoření cesty pomocí objektu `RouteBuilder`, který musí být návratovým typem anotované metody. Metoda nesmí mít žádný formální argument, musí mít veřejný modifikátor přístupu a musí být virtuální. Pokud `@UseRouteBuilder` má hodnotu *true* (což platí, pokud není explicitně nastavena na *false*), musí být v testovací třídě uvedena alespoň jedna metoda anotovaná touto anotací.

Pro účely testování lze existující cesty měnit i po jejich vytvoření a zaregistrovaní v kontextu. Tato práce se limituje pouze na API v jazyce Java, ale jak bylo řečeno, Camel je možné použít i v jiných jazycích kompilovaných do Java byte kódu a cesty je možné definovat i pomocí XML. Díky tomuto propojení je možné pomocí JUnit testovat Camel aplikace, které nebyly napsány v Javě. Pokud nastane potřeba dočasně změnit některou z cest, lze použít anotaci:

- `@UseAdviceWith` – slouží k anotaci třídy a má jeden argument typu `boolean`, jehož výchozí hodnota je *true*. Pokud je tato funkcionalita povolena, lze si z *CamelContext* načíst jakoukoliv z registrovaných cest a modifikovat ji. Při použití této anotace je třeba kontext ručně spustit a poté vypnout – změnu cesty není možné provést za běhu kontextu.

CamelContext je ve výchozím nastavení vytvářen zvlášť pro každou testovací metodu a po provedení poslední metody anotované `@After` je odstraněn, pro běh další testovací metody je pak vytvořen znova. Tím je zajištěna isolace testů, nicméně existují testovací případy, které pracují pouze s jednou instancí kontextu, ale například z důvodu čitelnosti byly rozděleny do více metod. Proto je možné toto chování modifikovat anotací:

- `@CreateCamelContextPerClass` – slouží k anotaci třídy a má jeden argument typu `boolean`, jehož výchozí hodnota je *true*.

Jakmile je spuštěn test a do systému odeslána zpráva, spustí se časomíra, jak dlouho čekat na doručení zprávy. Mohla nastat chyba a zpráva nebyla doručena, v takovém případě by se program nikdy neukončil, s tím testovací prostředí počítá a po určité době test ukončí. Tato doba je defaultně nastavena na 10 sekund, lze ji změnit anotací:

- `@ShutdownTimeout` – lze s ní anotovat pouze třídu, má dva argumenty: `value` typu `int`, který určuje dobu, jak dlouho čekat a `timeUnit`, což určuje jednotku definovanou výčtovým typem `java.util.concurrent.TimeUnit`.

V kapitole o projektu Camel byla představena možnost nahradit skutečné koncové body virtuálními mocky. K tomu slouží dvě anotace, které mají jeden argument typu `String`. Ten představuje koncový bod, určený pro nahrazení virtuální komponentou. Lze také použít znak * a tím určit větší počet koncových bodů určených pro nahrazení. Například výraz `log*` nahrazuje všechny koncové body, jejichž název začíná řetězcem *log*:

- `@MockEndpoints` – skutečná komponenta není vytvořena, ale nahrazena mock komponentou, kterou lze v testu využít a posílat jí data.
- `@MockEndpointsAndSkip` – místo skutečné komponenty je vytvořena pouze mock komponenta, která není zahrnuta do testu. Dá se říci, že je nahrazena prázdnou operací a když na ni přijde při testu řada, ihned se pokračuje dále.

Na procesory nacházející se v cestě je možné uplatnit jednoduché ladící operace pomocí anotací:

- `@UseDebugger` – touto anotací lze povolit použití debuggeru. Má jeden argument typu `boolean`, defaultní hodnota je *true*. Lze použít pouze pro anotaci třídy.
- `@ProvidesBreakpoint` – slouží k definici operací, které se mají stát při zpracování procesoru. Touto anotací lze anotovat pouze metody, které mají veřejný modifikátor přístupu, jsou virtuální a jejich návratová hodnota je typu `Breakpoint`, což je rozhraní deklarující dvě metody. Jedna určuje ladící operace, které se mají vykonat před zpracováním procesoru a ta druhá operace po zpracování procesoru.

V testu lze nastavit enviromentální proměnné (jednoduché úložiště typu klíč-hodnota), které se zaregistrují do registru kontextu. Pokud není použito OSGi nebo framework Spring (k čemuž tento *test runner* neslouží), je použit JNDI[1] registr. Jelikož se v tomto případě jedná o ukládání hodnot typu `Object`[2] s klíčem stejného typu, je jako implementace JNDI použita jednoduchá mapa `Hashable<Object, Object>`.

- `@CreateRegistry` – slouží pro přidání párů klíč-hodnota do registru kontextu. Lze s ní anotovat pouze metody s veřejným modifikátorem přístupu, které jsou virtuální a jejich návratový typ je `JndiRegistry`. Jelikož registr je vytvářen kontextem a uživatel do něj pouze přidává nové hodnoty, musí metoda mít právě jeden formální argument typu `JndiRegistry`, uživatel do něj přidá nové hodnoty a poté jej navrátí kontextu.

Camel aplikace je možné monitorovat a spravovat pomocí JMX[3]. Ve výchozím nastavení je tato funkcionalita vypnuta a pro zapnutí slouží anotace:

- `@DisableJMX` – definuje jeden argument typu *boolean* a ten má ve výchozím stavu hodnotu *true*. Pro povolení JMX je tedy nutno zadat hodnotu *false*. S touto anotací lze anotovat pouze třídy.

Výčet anotací dokončují poslední dvě anotace věnující se nastavení PropertyComponent, což je způsob, jakým Camel od verze 2.3 může vyhledávat třídy vyskytující se v `RouteBuilder` výrazech:

- `@UseOverridePropertiesWithPropertiesComponent` – redefinice nastavení pro účely testování. Lze anotovat pouze třídy.
- `@IgnoreMissingLocation` – ignorovat chybový stav, kdy třída použitá jako koncový bod není nalezena. Lze anotovat pouze třídy.

[1] Java Naming and Directory Interface – rozhraní pro přístup k adresářové službě. Popisuje pouze rozhraní, nikoli implementaci. Existují implementace pro práci s LDAP, databází či souborovým systémem a mnoho dalších.

[2] Narozdíl od C++ je v Javě použit objektový model s kořenovým objektem – Object

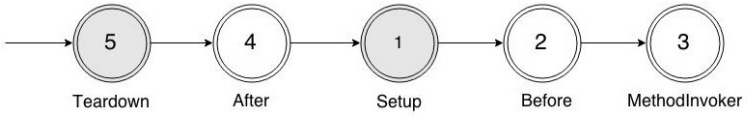
[3] *Java Management Extensions* – nástroje pro monitorování a správu Java aplikací.

3.5 Implementace

Na začátku této práce byly popsány anotace a způsoby, jakými s nimi lze pracovat. Mělo by tedy být zřejmé, že výše uvedené anotace je třeba zkompilovat do byte kódu a za běhu programu je vyhledávat. *Retention* všech anotací tak je *runtime*, zároveň jsou zahrnuty do dokumentace a je možné je dědit (to spíše z důvodů jisté benevolence ke kreativitě a potřebám testera, obecně rozsáhlé testy s dědičností nejsou praktické).

Pro vyhledání anotací v byte kódu je použita knihovna *reflection* – navrhnutý *test runner* prohledává pouze jeden soubor, což je dostatečný limit paměťové náročnosti.

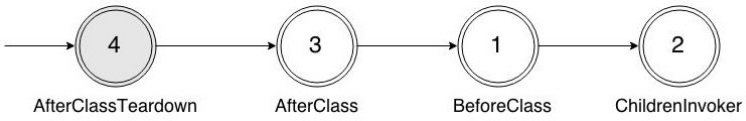
V příloze A je diagram tříd, na kterém lze vidět rozšíření výchozí *test runner* třídy o podporu Apache Camel. Nový *test runner* je pojmenován `CamelRunner` a dědí od třídy `BlockJUnit4ClassRunner`. Bylo také již zmíněno, které metody je třeba redefinovat, pro úpravu dle potřeb projektu Camel. Na úrovní metod je přidána inicializace kontextu a testovacího prostředí, po dokončení činností na úrovni metody je proveden pouze restart vnitřních hodin. Na následujícím diagramu jsou tyto operace vyobrazeny, standardní operace JUnit jsou vyznačeny světle, nově přidané operace jsou vyznačeny tmavě.



Obrázek 3.1: *Operace na úrovni metody (čísla označují, v jakém pořadí jsou provedeny)*

Řídící algoritmus nejprve navštíví událost *Teardown*, ta ale volá metodu `evaluate` události *After*, která rovněž nemá dostatečnou prioritu pro spuštění. Až událost *Setup* je provedena, kde se nachází inicializace kontextu. Poté jsou provedeny události zpracovávající `@Before` metody a `@Test` metody. Nyní se řízení vrací k události *After*, kterou je již možné provést a nakonec je proveden úklid. Tím končí jeden cyklus algoritmu řízení testování na úrovni metod.

Na úrovni třídy se nenachází žádná příprava, což je jediný okamžik, kdy v testu ještě neexistuje kontext. Nachází se zde však ukončovací metoda pro dokončení vláken. Algoritmus je založen na stejném principu. Událost *ChildrenInvoker* spouští operace na úrovni metod pro každou metodu v testovací třídě anotovanou `@Test`.



Obrázek 3.2: *Operace na úrovni třídy*

Kapitola 4

Srovnání definic testů

Způsob, jakým jsou Camel aplikace testovány, je touto prací zásadně změněn. Původní iniciativa byla pouze přidat nové anotace, výsledná implementace však přinesla zcela nový *test runner* určený konkrétně pro Camel.

Tato kapitola prakticky představí nové API, založené na anotacích, a porovná jej s API definovaném v `CamelTestSupport`. Budou zde použity dvě anotace, které byly definovány komunitou již před započetím prací na novém API a přinášejí do testu velmi důležité vlastnosti pomocí dependency injection. Proto byly přidány i do nového API testovací komponenty. U obou anotací se jedná o anotování instančních proměnných:

- `@Produce` – testování Camel aplikace fakticky znamená zasílat zprávy do systému a kontrolovat stavy koncových bodů. Touto anotací lze vytvořit zdroj, produkující zprávy do systému. Tato anotace má celkem 4 argumenty, nejdůležitější je argument `uri`, který určuje koncový bod, ze kterého je produkována zpráva dále do systému.
- `@EndpointInject` – jelikož testovací třída již nedědí od žádné Camel třídy, nejsou ji přímo zpřístupněny metody, na kterých je staré API založeno. Pro zpřístupnění *MockEndpoint* komponenty slouží tato anotace, vytváření konkrétního *mock* koncového bodu již není součástí testu, ale děje se nezávisle a navíc deklarativním způsobem.

4.1 Příklad vytvoření jediného kontextu v rámci testovací třídy

Jelikož nových anotací je poměrně mnoho a představování každé na jednotlivém příkladu by bylo poměrně vyčerpávající, jsou prezentovány pouze některé. Následující kód prezentuje rozdíl v definici testu, který spouští pouze jeden kontext pro celou testovací třídu. Bez anotací:

```
public class TestCase extends CamelTestSupport {

  @Override
  public boolean isCreateCamelContextPerClass() {
    // Má být CamelContext vytvořen pouze jednou?
    // Pouze jedinný řádek
    return true;
  }

  @Test
  public void test1() {
    // Tělo testu
```

```
  }


  @Test
  public void test2() {
    // Tělo testu
  }

  @Override
  protected RouteBuilder createRouteBuilder() {
    // Vytvoření objektu RouteBuilder
  }
}
```

Příklad 4.1: *Pouze jeden kontext, bez anotací*

Lze vidět, že testovací třída dědí od třídy `CamelTestSupport` a je tak na ní závislá. Metody anotované `@Override` jsou redefinice metod z `CamelTestSupport`. Test s anotacemi:

```
@RunWith(CamelRunner.class)
@CreateCamelContextPerClass
public class TestCase {

  @Test
  public void test1() {
    // Tělo testu
  }

  @Test
  public void test2() {
    // Tělo testu
  }

  @CreateRouteBuilder
  public RouteBuilder createRouteBuilder() {
    // Vytvoření objektu RouteBuilder
  }
}
```

Příklad 4.2: *Pouze jeden kontext, s anotacemi*

V tomto testu již testovací třída není závislá na žádné jiné třídě. *Test runner* je určen argumentem anotace `RunWith`, vytvoření pouze jednoho kontextu je rovněž určeno anotací. Další anotace je použita pro vytvoření cest pomocí objektu `RouteBuilder`.

4.2 Příklad nahrazení závislého objektu abstraktním objektem

Následující test demonstruje použití anotace `@IsMockEndpoints` a připravení testovacího prostředí pomocí anotace `@Before`. Tato ukázka již není zkrácena, uvedený kód lze přeložit a spustit. Jako první je opět uvedeno řešení bez použití anotací, kdy jakákoli úprava je prováděna redefinicí některé z metod třídy `CamelTestSupport`. I když je možné pro přípravu testovacího prostředí použít anotace JUnit, je demonstrována další možnost využitím metody `setUp` definované ve třídě `CamelTestSupport`. Pomocí třídy `ProducerTemplate`, jejíž instance je v API bez anotací přístupná skrze dědičnost, je vytvořena zpráva a ta je následně zaslána do systému. V testu 4.3 jsou takto vytvořeny dvě zprávy, každá s jinou hlavičkou a přenášeným souborem. Kód bez anotací:

```
public class IsMockEndpointsFileTest extends CamelTestSupport {

  @Override
  public void setUp() throws Exception {
    deleteDirectory("target/input");
    deleteDirectory("target/messages");
    super.setUp();
  }

  @Override
  public String isMockEndpoints() {
    // Koncové body dle tohoto vzoru budou zastoupeny mock objekty.
    return "file:target*";
  }

  @Test
  public void testMockFileEndpoints() throws Exception {
    // Získání referencí na mock objekty
    MockEndpoint camel = getMockEndpoint("mock:file:target/messages/camel");
    MockEndpoint other = getMockEndpoint("mock:file:target/messages/others");

    // Nastavení očekávání
    camel.expectedMessageCount(1);
    other.expectedMessageCount(1);

    // Vytvoření a odeslání zpráv
    template.sendBodyAndHeader("file:target/input", "Hello Camel",
      Exchange.FILE_NAME, "camel.txt");
    template.sendBodyAndHeader("file:target/input", "Hello World",
      Exchange.FILE_NAME, "world.txt");

    // Vyhodnocení testu
    assertMockEndpointsSatisfied();
  }

  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
    // Vytvoření cesty s content-based router
    return new RouteBuilder() {
      @Override
      public void configure() throws Exception {
        from("file:target/input")
          .choice()
            .when(body(String.class).contains("Camel"))
            .to("file:target/messages/camel")
            .otherwise().to("file:target/messages/others");
      }
    };
  }
}
```

Příklad 4.3: *MockEndpoints příklad, bez anotací*

Test využívající anotace je založen na stejných principech, požadovaných cílů je však dosaženo způsobem, kdy vytvoření závislostí potřebných pro test je provedeno deklarativním způsobem a je tak odděleno od samotného testu. V metodě anotované `@Test` se tak nachází pouze kód vykonávající akce popsané v kapitole o testování Camel aplikací – nastavení očekávání, zaslání zpráv a vyhodnocení, zda očekávání byla splněna.

```
@RunWith(CamelRunner.class)
@MockEndpoints("file:target*")
public class IsMockEndpointsFileTest {

  @ContextInject
  public CamelContext context;

  @EndpointInject(uri = "mock:file:target/messages/camel")
  public MockEndpoint camelEndpoint;

  @EndpointInject(uri = "mock:file:target/messages/others")
  public MockEndpoint othersEndpoint;

  @Produce(uri = "direct:start")
  public ProducerTemplate template;

  @Before
  public void setUp() throws Exception {
      deleteDirectory("target/input");
      deleteDirectory("target/messages");
  }

  @Test
  public void testMockFileEndpoints() throws Exception {
    camelEndpoint.expectedMessageCount(1);
    othersEndpoint.expectedMessageCount(1);

    template.sendBodyAndHeader("file:target/input", "Hello Camel",
      Exchange.FILE_NAME, "camel.txt");
    template.sendBodyAndHeader("file:target/input", "Hello World",
      Exchange.FILE_NAME, "world.txt");

    camelEndpoint.assertIsSatisfied();
    othersEndpoint.assertIsSatisfied();
  }

  @CreateRouteBuilder
  public RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
      // Definice cesty je shodná s předchozím příkladem
    };
  }
}
```

Příklad 4.4: *MockEndpoints příklad, s anotacemi*

Jak již bylo řečeno, obě implementace dosahují stejných cílů, avšak nezačleňování *test fixtures* akcí do testovacích metod zvyšuje čitelnost a udržovatelnost testů. V obou příkladech jsou v rámci přípravy testovacího prostření odstraněny adresáře *input* a *messages* – vyprázdnění adresáře *target*. Pomocí objektu `RouteBuilder` je vytvořena cesta implementující vzor EIP *content-based router* – z adresáře *input* je odeslána zpráva a systémem je směrována dle následujících pravidel: Pokud tělo zprávy obsahuje řetězec *Camel*, zpráva je uložena do adresáře *camel*, jinak je uložena do adresáře *others*. Adresář *input* však není nikdy v souborovém systému vytvořen, dle vzoru uvedeném v metodě `isMockEndpoints` je nahrazen *mock* objektem. V testu jsou pak nastavena očekávání, odeslány zprávy do systému a následně vyhodnocena očekávání, zda byla splněna.

Kapitola 5

Závěr

Cílem této práce bylo analyzovat testovací komponentu projektu Apache Camel a identifikovat imperativní kód, vhodný pro nahrazení anotacemi. Navrhnout anotace a způsob, jakým lze anotace nalézt pro jejich zpracování během běhu programu. V testovací komponentě bylo nalezeno celkem třináct metod, které je vhodné zapisovat anotacemi namísto volání metod a čtrnáctá anotace byla navržena pro zpřístupnění samotného jádra Apache Camel (*CamelContext*) testovací třídě pomocí dependency injection.

Všechny anotace byly implementovány pomocí standardních prostředků v rámci Java SE a pro práci s anotacemi byla použita knihovna *Reflection*. Pro nalezení anotace třídy je třeba navštívit všechny třídy a poté všechny anotace dané třídy, což je algoritmus s kvadratickou asymptotickou složitostí. Pro nalezení anotací metod a proměnných je třeba v každé třídě navštívit každou metodu a všechny anotace u každé metody, resp. proměnné, což je algoritmus s kubickou asymptotickou složitostí, což pro programy složené z velkého počtu tříd je nevhodně pomalé. Proto bylo rozhodnuto kromě implementace anotací implementovat novou *test runner* třídu pro testovací framework JUnit, která odpovídá požadavkům projektu Camel i současným praktikám projektu JUnit ve verzi 4.12. Díky tomu odpadl požadavek vyhledávat testovací třídu, ta je *test runner* třídě předávána jako skutečný parametr při spuštění testu. Tím byla práce rozšířena na modernizaci testovací komponenty, která byla původně napsána podle JUnit verze 4.0.

Přínos této práce je zjednodušení a zpřehlednění definic testů projektu Apache Camel a nahrazení silných vazeb mezi objekty testu, které vznikaly dědičností objektů, vazbami slabými, kterých je dosaženo použitím vzoru depenency injection.

Literatura

[1] Allen, D.: Getting started [online]. http://arquillian.org/guides/getting_started, [cit. 2015-01-20].

[2] Beck, K.: What is the difference between unit testing, functional testing, and integration testing? [online]. http://www.quora.com/What-is-the-difference-between-unit-testing-functional-testing-and-integration-testing, [cit. 2015-01-20].

[3] CERT: Do not use reflection to increase accessibility of classes, methods, or fields [online]. https://www.securecoding.cert.org/confluence/display/java/SEC05-J.+Do+not+use+reflection+to+increase+accessibility+of+classes,+methods,+or+fields, [cit. 2015-01-20].

[4] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, iSBN-13: 978-0201633610.

[5] Google: The JRE Class White List [online]. https://cloud.google.com/appengine/docs/java/jrewhitelist?csw=1, [cit. 2015-01-20].

[6] Hohpe, G.; Woolf, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003, iSBN-13: 078-5342200683.

[7] Ibsen, C.; Anstey, J.: *Camel in Action*. Manning Publications, 2011, iSBN-13: 978-1935182368.

[8] Knutsen, A.; Rubinger, A. L.: *Continuous Enterprise Development in Java*. O'Reilly Media, 2011, iSBN-13: 978-1449328290.

[9] Oracle: Java SE Documentation, Annotations [online]. http://docs.oracle.com/javase/7/docs/technotes/guides/language/annotations.html, [cit. 2015-01-20].

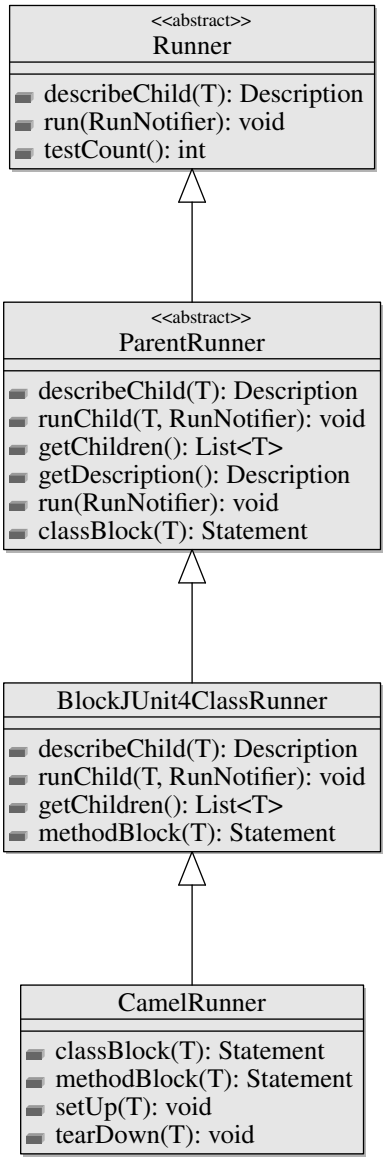
[10] OW2: ASM bytecode analysis framework [online]. http://asm.ow2.org/, [cit. 2015-01-20].

[11] Shigeru, C.: Java Programming Assistant [online]. http://www.csg.ci.i.u-tokyo.ac.jp/ chiba/javassist, [cit. 2015-01-20].

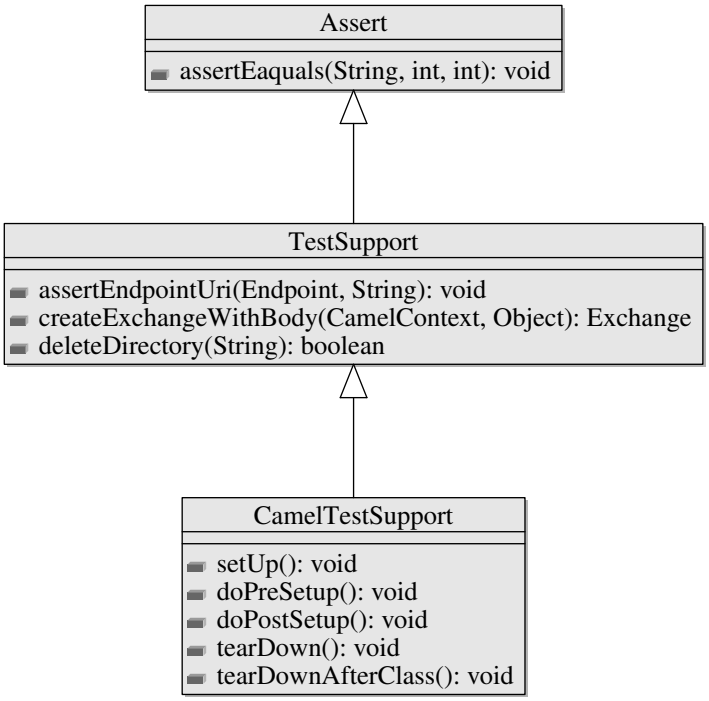
[12] Wellmann, H.: Configuration Options [online]. https://ops4j1.jira.com/wiki/display/PAXEXAM3/Configuration+Options, [cit. 2015-01-20].

Příloha A

Diagramy tříd



Obrázek A.1: *Camel test runner*



Obrázek A.2: *Architektura balíku camel-test (metody vynechány)*

Příloha B

Obsah přiloženého CD

- PDF bakalářské práce
- Zdrojové kódy bakalářské práce v LaTeX
- Zdrojové kódy komponenty *camel-test*
- Soubor readme.md s instrukcemi pro přeložení komponenty